Informing the Design of Pipeline-Based Software Visualisations

Neville Churcher **Warwick Irwin**

Software Engineering & Visualistion Group, Department of Computer Science & Software Engineering,
University of Canterbury, Private Bag 4800, Christchurch, New Zealand
E-mail: {neville,wal}@cosc.canterbury.ac.nz

Abstract

In this paper, we consider the process by which an effective software visualisation can be designed and explore the ways in which both special-purpose and general-purpose tools may be used to inform the software visualisation design process. A series of decisions must be made in order to determine which data will contribute, the 'look & feel' of the visualisation, the algorithms, stylesheets and configuration parameters which are involved as implementation progresses. In our previous work we have developed a flexible, extensible and configurable pipeline-based approach to the implementation of software visualisation. Data is represented in XML at each stage and undergoes successive transformations as it moves through the implementation pipeline. Pipeline components capture and analyse data, compute geometry and determine the detailed presentation of visual output. In this paper, we describe a parallel pipeline for software visualisation design. Its steps involve making choices which determine the specific implementation pipeline components, together with their configurations, defining a particular visualisation. We discus issues and techniques involved in the software visualisation design pipeline, describe tools which support them, and give examples from our software visualisation research.

1 Introduction

Software engineering typically involves the design, development and evolution of large, complex software systems. It is not uncommon for such systems to involve millions of lines of code (MLOC). Governments and businesses have made huge investments of capital, time and resources in these systems and their maintenance. Our reliance on banking, medical, control and other software systems is such that failure or flaws can be catastrophic.

Accepted practices for software quality assurance include the use of Fagan inspections (Fagan 1976, Fagan 1986) in which groups of software engineers read & study code listings and then meet to conduct reviews in order to detect and prevent defects. Typical inspection guidelines indicate that a rate of 100–125 LOC/hr should be allowed for code reading (O'Regan 2002). Thus, reading 1 MLOC at 125 LOC/hr would take 8,000 hours (200 40hr weeks, 4 staff years). The source for Windows 3.1 (1990) was around 3MLOC, current OS are around 10 times that size[1] (Freeman 1999), and would more than a century to read—even if they weren't continually being changed.

Traditional approaches struggle to cope with such systems. A natural way to address issues of scale and complexity is to divide the load among a larger team of people. While this reduces the individual load, it introduces additional problems of communication and coordination. Sadly, the evidence suggests that large projects, and large teams, are most likely to fail.

In order to make genuine advances, software engineers must be able to step back and take a "bird's eye" view of software. Rather than trying to comprehend all of the low level detail, it is necessary to be able to scan the entire system for likely trouble spots and then focus more on those areas which require closer attention. This *focus + context* concept is familiar in information visualisation and is the basis of many well-known techniques such as fisheye views (Furnas 1986, Sarkar & Brown 1994)

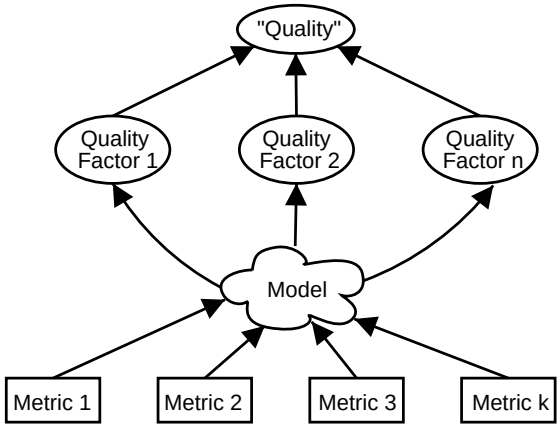

Figure 1: Quality models, metrics and quality

Software metrics research has concentrated on identifying quantitative measures of specific properties of software, together with models which relate these to aspects of interest (Fenton & Pfleeger 1997, Ince & Shepperd 1992, Henderson-Sellers 1996, for example). The archetypal example is Lines of Code (LOC), which has become the *de facto* measure of software size but has also been used with varying success as an predictor of other attributes such as defect counts. Another example is cyclomatic complexity (ν) (McCabe 1976), which is used as a measure of complexity, which in turn is used as a quality factor indicating the likelihood of maintenance problems.

Software quality models, illustrated schematically in Figure 1, relate measurable software metrics to more abstract quality factors. A good model should

[1] http://www.campusprogram.com/reference/en/wikipedia/s/so/source_lines_of_code.html

be based on a sound theoretical foundation, be practical to implement and be capable of making testable predictions.

Early, yet long-lived, models such as that of McCall et al. (McCall, Richards & Walters 1977) (shown in Figure 2), were based on hierarchical decomposition into more specific quality indicators.

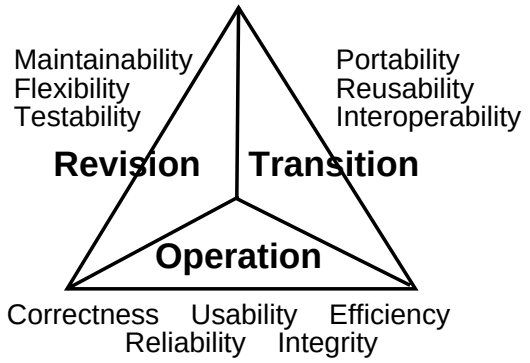

Figure 2: McCall *et al.* quality model

The software metrics discipline has enjoyed considerable success. However, a number of limitations have emerged. One serious limitation has been the difficulty of obtaining genuinely complete and accurate metrics; the effectiveness of any visualisation ultimately depends on the quality of the data upon which it is based. In previous work, we have developed robust metrics tools (Irwin & Churcher 2003) which are capable of delivering data of the required quality.

Substituting the more tractable analysis of software metrics for the impractical direct analysis of software artifacts is an enormous simplification. However, for larger systems and realistic numbers of metrics, even the reduced problem is daunting: the user is overwhelmed with data and it is hard to distinguish the features of interest.

Software visualisation has emerged as a means of addressing information overload challenges as well as supporting exploratory analysis and offering insight into the underlying science. Visualisations provide a way of connecting metrics data to views of the software artifacts, bridging the gap between metrics and direct analysis.

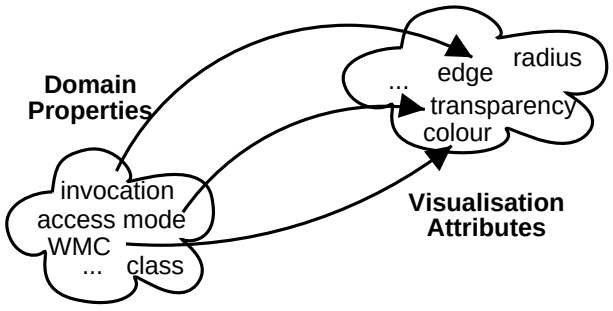

Figure 3: Visualisation via mappings

Figure 3 illustrates the process of software visualisation in general terms. Domain properties represent the components, connections and details of a software system. Visualisation attributes make up the view ultimately presented to the user. Essentially, a visualisation is determined by defining a set of mappings from a subset of the domain properties to a subset of the visualisation attributes. It is important to note that the domain properties may include both 'direct' properties, such as the presence of an inheritance relationship, or 'indirect' properties, such as software metrics like WMC (Chidamber & Kemerer 1994), which require additional computation to derive from the software artifacts.

Typically, the mapping process shown in Figure 3 is decomposed into a series of steps. This pipeline-based approach is discussed further in the following section.

In this paper, we consider the process by which an effective software visualisation can be designed. A series of decisions must be made in order to determine which data will contribute, the 'look & feel' of the visualisation, the algorithms, stylesheets and configuration parameters which are involved as implementation progresses. We describe these decisions in terms of a design pipeline, which parallels the implementation pipeline. Our goal is to explore the ways in which both special-purpose and general-purpose tools may be used to inform the software visualisation design process.

The remainder of the paper is structured as follows. In the next section, we describe the pipeline-based approach to implementing software visualisations and develop the concept of the parallel design pipeline which accompanies it. In Section 3 we describe SeeSoftLike, a tool we have developed to help in the design process, and discuss some of the issues arising in its use. The use of a more general-purpose tool is discussed in Section 4. In Section 5 we consider the rôle of the design pipeline in class cluster visualisation. Our conclusions, together with a description of our continuing work, are presented in Section 6.

2 Pipeline-based visualisation

The conventional visualisation pipeline (Schroeder, Martin & Lorensen 1998, for example) consists of three basic phases: data capture/generation & processing; computation of geometry; rendering of the resulting visualisation.

Our recent work in information and software visualisation (Churcher, Keown & Irwin 1999, Irwin & Churcher 2002, Churcher, Irwin & Kriz 2003, Irwin & Churcher 2003) involves a flexible extensible pipeline approach to information and software visualisation.

The pipeline is depicted, in simplified form, at the left of Figure 4 and the major steps are summarised here. We will refer to this as the *implementation pipeline* to distinguish it from the *design pipeline* which will be developed later in this section.

- Our data capture tools are developed with the aid of yakyacc, an XML-based parser generator, and JST, a Java semantic model builder (Irwin & Churcher 2003). These enable us to obtain, in terms of the corresponding standard grammar for the programming language in question, values for arbitrary software metrics. Typically, this involves performing XSLT transformations on the XML representations of the semantic model and corresponding parse trees.

  Our approach is sufficiently flexible that we can accept input from other tools, provided that it can be transformed appropriately into our XML format. For example, we also use the metrics module of Borland's Together IDE (together 2004) as a source of data.

- In general terms, the data obtained represents components and connections or relationships between them. Both components and relationships may have further detailed properties. For example, a method, which has a return type and parameter list, may override another method.

Pre-layout filters determine which of the available components and relationships, as well as which of their properties, will participate in the generation of a particular visualisation. For example, only public methods might be included and inheritance relationships between classes might be omitted.

- Geometry can now be computed, based on the filtered data. For example, this might involve the use of ANGLE (Churcher & Creek 2001) to perform 3D graph layout where the filtered data is in the form of nodes and edges.
- Post-layout filters determine the details of the geometry which will be presented to users: sizes, shapes, colours, fonts, transparencies and other visual attributes are specified at this stage.
- Finally, the resulting visualisation is tailored for a particular presentation tool or context.

XML is used to represent the data at each stage, and filters are typically implemented as XSLT transformations. This approach provides considerable flexibility and enables ready validation at each stage.

We refer the reader to our other publications for further detail of implementation pipeline components and examples of the application of this approach.

In practice, the process of developing and realising visualisations involves two parallel pipelines. One, as described above, essentially represents the automated development of the visualisation. The second, implicit, pipeline—depicted on the right of Figure 4—consists of the successive design decisions which are embodied in the implementation pipeline.

The correspondence between the stages of the implementation and design pipelines is not $1:1$. Some of the major correspondences between the stages of the design and implementation pipelines are summarised in Table 1.

Design	Implementation
Metrics selection	data capture and analysis tools
Semantic relationships	Data analysis, pre-layout filters
Metaphor selection	Pre-layout filters, geometry computation
Design	algorithms, techniques, geometry computation
Presentation	Post-layout filters, customisation transformations
Configuration	XSLT, ...

Table 1: Correspondences between design and implementation pipelines

In this paper, we consider the design pipeline, and its relationships with the implementation pipeline.

We identify the following design pipeline stages:

Metrics selection is a particularly critical step in software visualisation. It is important that metrics be used which genuinely fit with the current quality model. Thus, it might not be appropriate to use LOC as a surrogate for complexity. A representative basis set should be chosen from a sufficient number of distinct categories (size, complexity, coupling, ...) to ensure that the visualisation will accurately reflect genuine and pertinent system features.

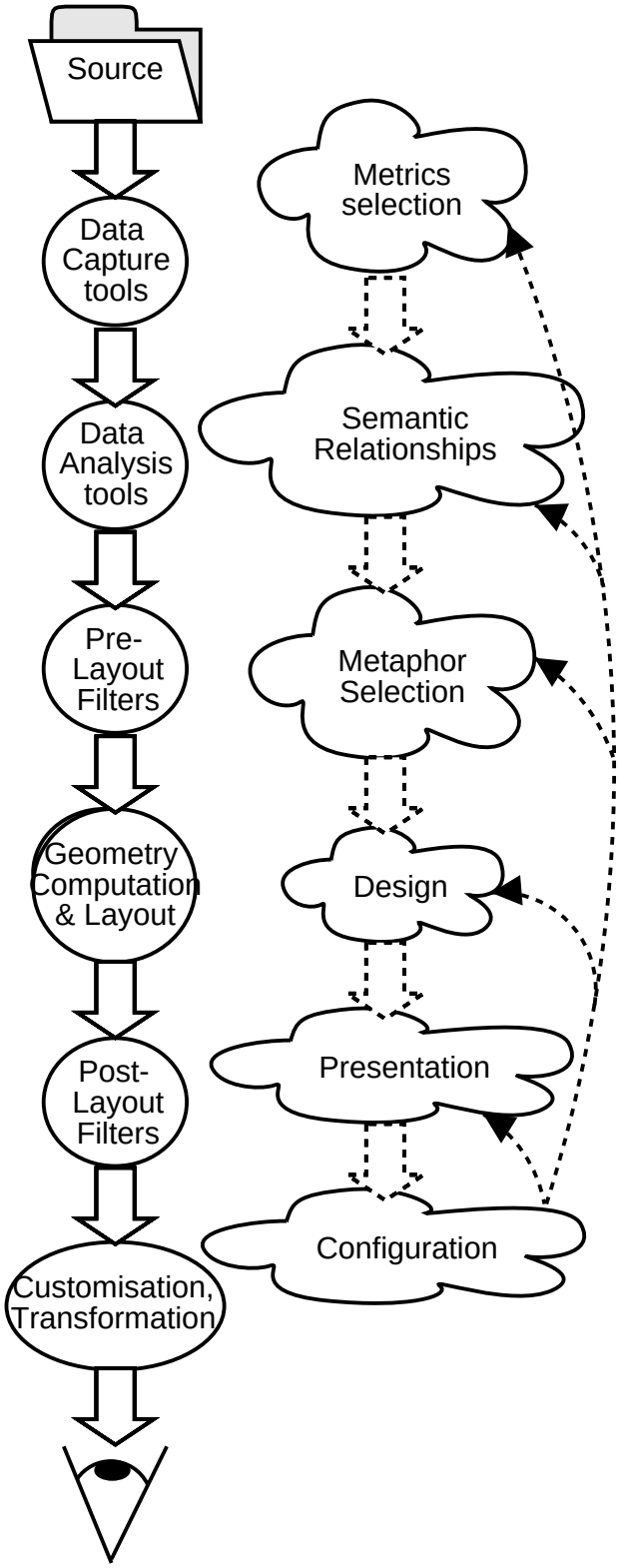

Figure 4: Parallel visualisation pipelines: implementation pipeline (left) and design pipeline (right)

In our case, this step typically involves determining which metrics to omit, since we may obtain almost any metric, directly or indirectly, from the semantic model and associated parse trees.

Semantic relationships arise in a generalised data model of object oriented software structure. Although their importance has been realised for a long time (Churcher & Shepperd 1995), the lack of powerful tools has prevented full advantage being taken of them. They include relatively straightforward relationships such as *class has method* or *class is ancestor of class*. However, accurate data about more subtle relationships, such as *method overrides method* cannot be obtained without a complete semantic model such as our JST.

As an example, in designing a specific visualisation, we might decide to preserve all inheritance relationships and omit relationships representing the implementation of interfaces.

Metaphor selection involves major decisions about dimensionality, perspectives, structures and relationships. These choices constrain the range of contexts in which a given visualisation will be useful and they are driven by the nature of the questions the visualisation is intended to help answer.

They should be based on domain-specific factors including the problems users will be addressing when they turn to software visualisations for assistance. Typically, these involve particular granularity of components (system, package, class, ...) and definition of a 'neighbourhood' of related components.

Typically, a metaphor will carry with it a geometric component as well as aspects specific to the software domain. These might include nested boxes, graphs and so on.

At this stage, it is useful to make use of a domain-specific taxonomy of visualisations, in order to assist with metaphor selection. A well known but dated taxonomy exists (Price, Baecker & Small 1993) but its usefulness is limited by its focus on visualisation systems/packages rather than the underlying visualisations themselves. A number of more generally-applicable software visualisation perspectives have been identified (Keown 2000). These affect aspects such as the user's perception of the geometry as either an object to manipulate or a world to observe, navigate and experience.

Two examples of common perspectives are:

- A hierarchy-based metaphor might be based on classes as major components and the inheritance relationships between classes as the dominant relationship. Other relationships, such as composition, between classes might be included in a secondary manner or be omitted altogether. Internal details of class structure might be omitted or presented only in an aggregated way.
- Alternatively, the metaphor might be based primarily on the interactions and relationships between methods (invocation, overloading, overriding, ...), with aspects relating to the classes to which the methods belong being of secondary importance.

Design Once a metaphor has been identified, the design stage involves choices about how it can be realised. For example, a hierarchical inheritance-centric metaphor might be realised via cone trees (Robertson, Mackinley & Card 1991), treemaps (Johnson & Shneiderman 1991) or some other technique. Such factors as layout algorithms will also be considered at this stage.

Presentation This phase involves choices about aspects such as the specific geometry to be used for nodes—it represents the targets of the mapping arrows in Figure 3.

Configuration Further processing may be required: for example, in order to obtain the data comprising the visualisation in a form suitable for display via a specific tool.

Both pipelines should be regarded as iterative, as suggested by the arrows shown for the design pipeline in Figure 4. This reflects the fact that information visualisation in general, and software visualisation in particular, is an exploratory process. Following the observation of features in one visualisation, a user is likely to customise some aspects, or select a related visualisation. Such behaviours are an important part of the problem solving or design tasks the user is addressing and it is necessary to accommodate them.

We contend that choices involved in the design pipeline stages can be informed by effective use of tools which highlight relevant features of the data sets. The next two sections describe examples of such tools and their use.

3 SeeSoftLike

In this section, we discuss SeeSoftLike, one of the tools we have developed for steering visualisation design. Our application, named after the system which first introduced this approach (Eick, Steffen & Jr. 1992), essentially displays source code in a microscopic font. The major features are still evident, as recognisable patterns of line lengths. The colours of individual lines represent the corresponding values of a variable whose domain has been specified as described below. This approach has been used for a variety of purposes (Ball & Eick 1996, Eick, Graves, Karr, Marron & Mockus 2001). Such applications typically obtain their data from version control systems such as CVS.

Our implementation is designed to be used either stand-alone or as part of our pipeline -based approach to information and software visualisation (Irwin & Churcher 2002). It reads XML data files which conform to a specified DTD. Domains are specified for attributes which are present in the data file and which are to be made available for colouring the lines in the basic file display.

Two kinds of simple domains may be defined for the variables used to colour lines. This is achieved by providing metadata including the corresponding variable name and the range of its values. *Nominal* domains, such as author, are defined by listing their values. *Numeric* domains, such as number of defects, are described in terms of lower and upper bounds. Other attributes may appear in the data file but only those for which domains have been defined are available for use in colouring the display.

Figure 5 shows a fragment of a data file—in this case one of the Java source files for the tool itself. The domains element contains definitions of a nominal and a numeric domain as described above. It is followed by a file element which may contain a sequence of block and line elements, where a block may, in turn, contain further blocks and lines.

```
<domains>
  <nominal name="author" values="carl neville wal"/>
  <numeric name="defects" min="0" max="100"/>
</domains>
<file name="DomDiddler" author="neville">
  <block name="getPanel" defects="25">
    <line>    public JPanel getPanel() {</line>
    <line author="wal">return jp;</line>
    <line>    }</line>
  </block>
</file>
```

Figure 5: Sample XML data

The attributes specified as domains may occur at the file, block or line level. For example, Figure 5 includes a single line with author "wal" inside a block which has inherited the value "neville" of its author attribute from the enclosing file. This nested approach allows local changes to be distinguished.

Domain metadata is used to generate the colour tables which map attribute values onto the corresponding colours used to display individual lines.

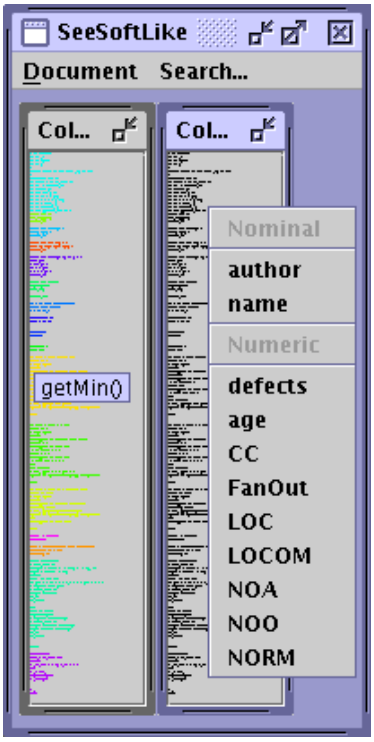

Figure 6: Using domains to colour file view

Figure 6 shows the basic tool interface. Two windows, each displaying the contents of the file `ColourTable.java` have been opened. A popup menu allows the user to select a domain attribute to be used to colour the lines of the file. The window on the right shows the initial view of the file while the window on the left shows the result of Selecting name from the popup menu. Blocks or lines are coloured according to the values of the name attribute and tooltips provide ready access to the specific values where required.

Figure 7 shows a typical scenario involving comparison of several files, each in its own window. The files are coloured according to their values of the *number of operations* (NOO) metric—the number of methods excluding inherited methods and constructors. NOO has only been specified at file level, so individual blocks and lines all appear in the same colour. The user has also displayed a colour table for

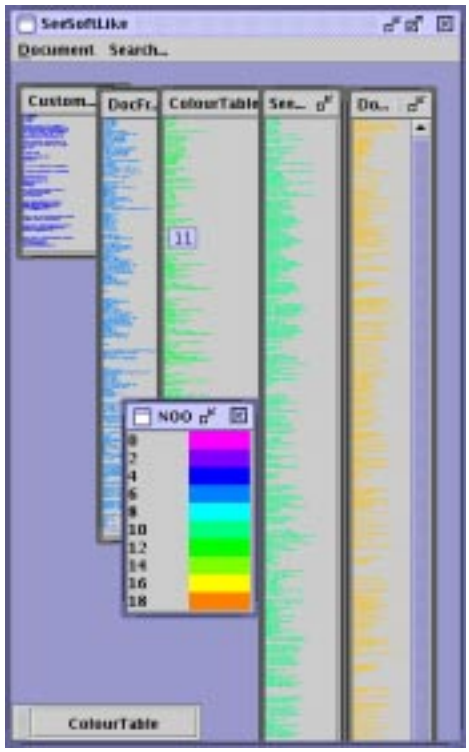

Figure 7: File comparison

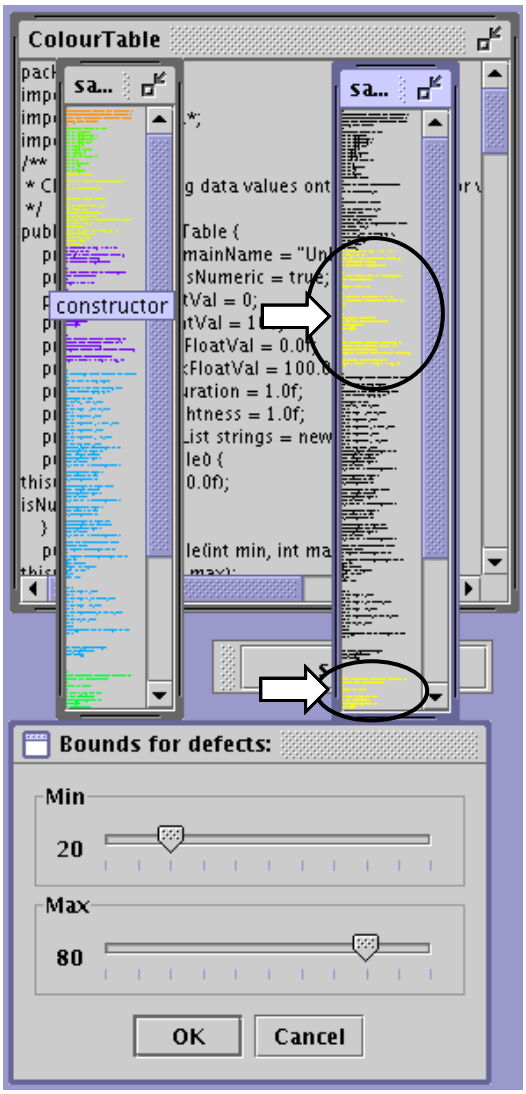

Figure 8: Filtering

this mapping in order to assist with interpretation of colour values. An iconified window, not involved in the current scenario, appears at the lower left. Even such relatively coarse visualisations can be useful. In this case the user is investigating the relationship between file length (LOC) and functionality (NOO).

A more detailed scenario is shown in Figure 8. The upper left window shows a Java source file coloured by a 'name' variable (corresponding to the name of a block containing a method, declaration section or import section). Lines whose values of an attribute fall within a specified range may be highlighted according to user-specified criteria. In Figure 8, the upper right window shows the same Java file, with highlighted regions (indicated by arrows) corresponding to regions where $0 \leq defects \leq 80$. The file windows may be toggled between the shrunken state and a readable size, as illustrated in the background ColourTable window in Figure 8.

Our SeeSoftLike tool supports a number of important aspects in the visualisation design process. Firstly, it provides a convenient way to scan simultaneously a large number of system artifacts in order to compare their values of individual attributes at file, block or line level of granularity. This helps us answer questions such as "which are the most complex methods" and "which authors seem to be associated with high defect density components?

Within a file, we can readily obtain an impression of the ranges and distributions of attribute values. This helps us to assess which metrics are sensitive to the features of the artifacts under consideration and which have relatively uniformly distributed values—and are thus less likely to be worth visualising. This allows us to tailor visualisations in order to emphasise interesting variations in values.

Comparison of multiple views of the same file gives an impression of the correlation of the corresponding attributes. Although arguably only semi-quantitative, this approach allows "interesting" features, such as low or negative correlation, to be detected. These features can then be investigated with more focussed tools.

This approach has some limitations, the major one being the lack of any explicit representation of connections (invocation, inheritance, composition and the like) between artifacts.

Our next version will be capable of highlighting regions of other open files which are related to selected regions of a target file: thus we will indicate the relationships by highlighting the related artifacts rather than through explicit connection geometry.

4 Using off-the shelf tools: Ggobi

Tools such as that described in Section 3 are valuable for identifying domain components and properties worthy of deeper investigation. Multivariate data visualisation software is available for general-purpose information visualisation and exploration. In this section, we describe how such tools can be used to inform our choices in the visualisation design pipeline.

The representative tool we will use to illustrate our arguments in this section is ggobi (ggobi 2004). It has an XML data format, which makes it amenable to our pipeline approach, provides a variety of visualisation options, can be integrated with other applications and has the desirable property of being freely available.

Data sets in software visualisation typically present a range of challenges including:

- High dimensionality; it would not be unusual to have tens or even hundreds of variables.
- High data volume, particularly where evolution over time is involved.
- Extreme ranges of values (perhaps several orders of magnitude).
- Distributions of individual metrics, such as LOC, are commonly skewed towards low values, but include a long tail.
- Outliers cannot simply be ignored as they are often indicate "interesting" features.

Figure 9 shows a snapshot of a ggobi session. Annotations are referred to in the following discussion.

A Ggobi provides several kinds of graph and the detailed content of the main configuration panel vary according to the type of the selected window. Common features include check boxes for selection of participating variables.

This is particularly useful when obtaining an overview of the ranges and distributions of metric values across a system in order to assess the presence of, and sensitivity to, various characteristics.

B Scatterplots, in various forms, are a powerful feature of ggobi. This example shows a 3D form, enabling correlations between several variables to be investigated. Other useful features include the ability to cycle through a series of scatterplots corresponding to each pair of variables.

The models underlying software metrics and software quality typically predict correlations between metrics and propose causal relationships to explain them. For example, a positive correlation between module complexity and defect density might be expected. Scatterplots allow expected correlations to be examined and unexpected ones to be detected and investigated further.

Often, families of metrics will effectively measure the same property of the software—for example, there are several forms of LOC depending on the way aspects such as comments, declarations and whitespace are treated. Normally, these metrics will be strongly correlated and a representative will be selected.

Other correlations are often expected and can readily be verified. For example, ν and LOC are generally strongly correlated, reflecting a relatively uniform number of decisions per unit length across a system. In this case, outliers are of interest and are readily observed.

C A scatterplot matrix permits the ready comparison of several 2-variable scatterplots at once. This is particularly useful in our case, where the greater complexity of the semantic model underlying object oriented software means that many more component kinds (packages, interfaces, ...) and relationships (inheritance, composition, ...) are involved than in the case of procedural languages. Each added dimension introduces additional metrics to be interpreted and managed. The matrix view enables trends of correlations between a variable and a family of others to be investigated conveniently.

D Parallel coordinates are a powerful tool for detecting similarities between components. Each line in the figure represents a single class and crosses the 5 vertical axes, each representing a class level metric, at the corresponding height. Trends, such as the low CC → high DOIH → low

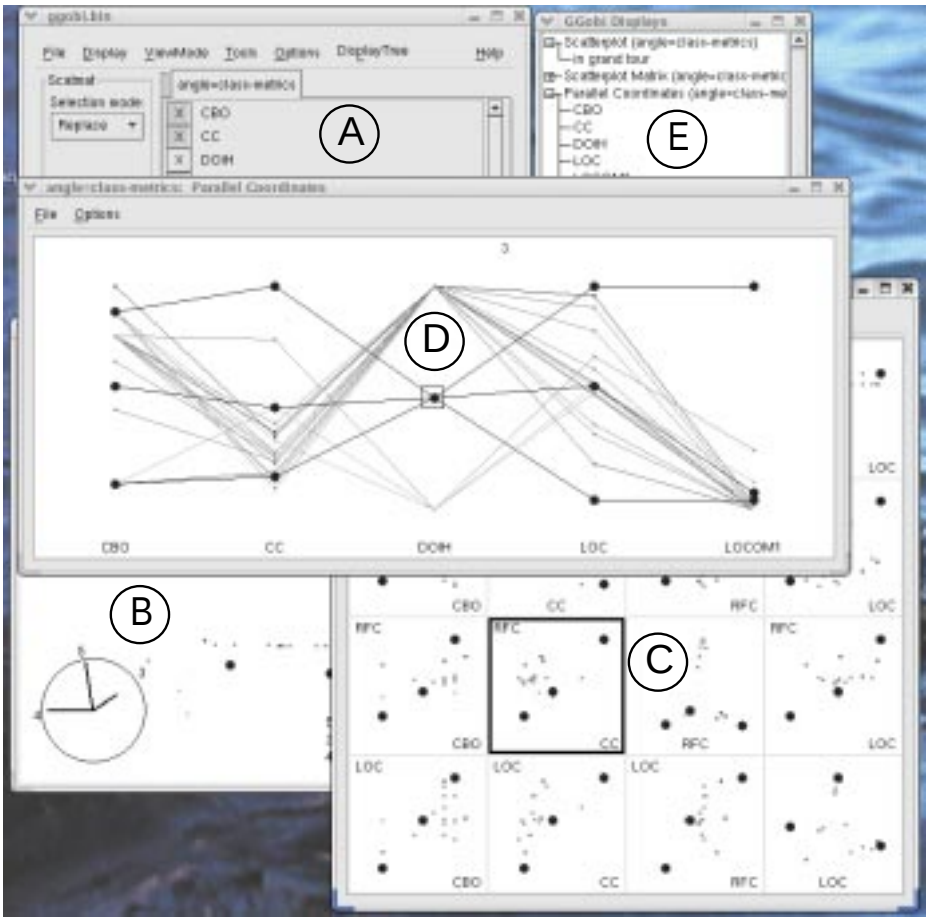

Figure 9: A ggobi session in progress

LOCOM1, indicating similarities between components are readily apparent (as are outliers). Features such as the ability to re-reorder the variables are very useful in the elimination of bogus effects. Similarity measures are particularly important in software visualisation, both for classification of components and for determining strongly related groupings. This has much in common with the concepts of statistical clustering (Everitt, Landau & Leese 2001).

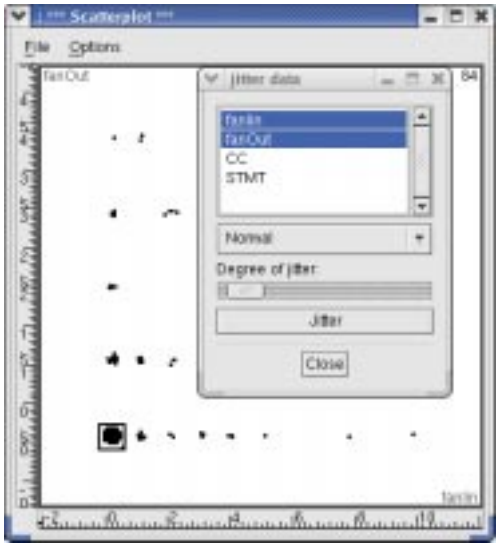

Figure 10: Use of jittering to distinguish co-located data points

E A tree structure simplifies the management of the many windows which are likely to be involved in a typical session.

Ggobi has a number of further features which are particularly useful, either in adding value to individual plots or in linking plots together. These include:

Colour schemes allow items, such as the glyphs on a scatterplot, to be coloured according to the value of some variable. This is particularly useful in situations such as colouring the points on a 2-variable scatterplot by their values of a third variable. This is useful in situations such as that shown in Figure 10: points on a plot of fan-in against fan-out might be coloured according to their ν values in order to provide a deeper understanding of the corresponding call graph.

Jittering enables co-located points to be resolved individually. Often the values of a metric will be limited to integer values and many points may correspond to the same values. Figure 10 shows a plot of fan-in (number of invoking methods) against fanout (number of invoked methods) for 152 methods. Many methods share identical pairs of values for these methods; the square at the lower left contains 84 points—more than half. This situation commonly arises because the distribution of values for many metrics is skewed towards low values. Jittering has been used to indicate by larger 'blobs' the presence of such degenerate points. Without it, important information about the invocation structure will be missed.

Brushing provides a way to select particular items and highlight them where they appear in other graphs. In Figure 9 the square at the centre of the parallel coordinate display (D) is the

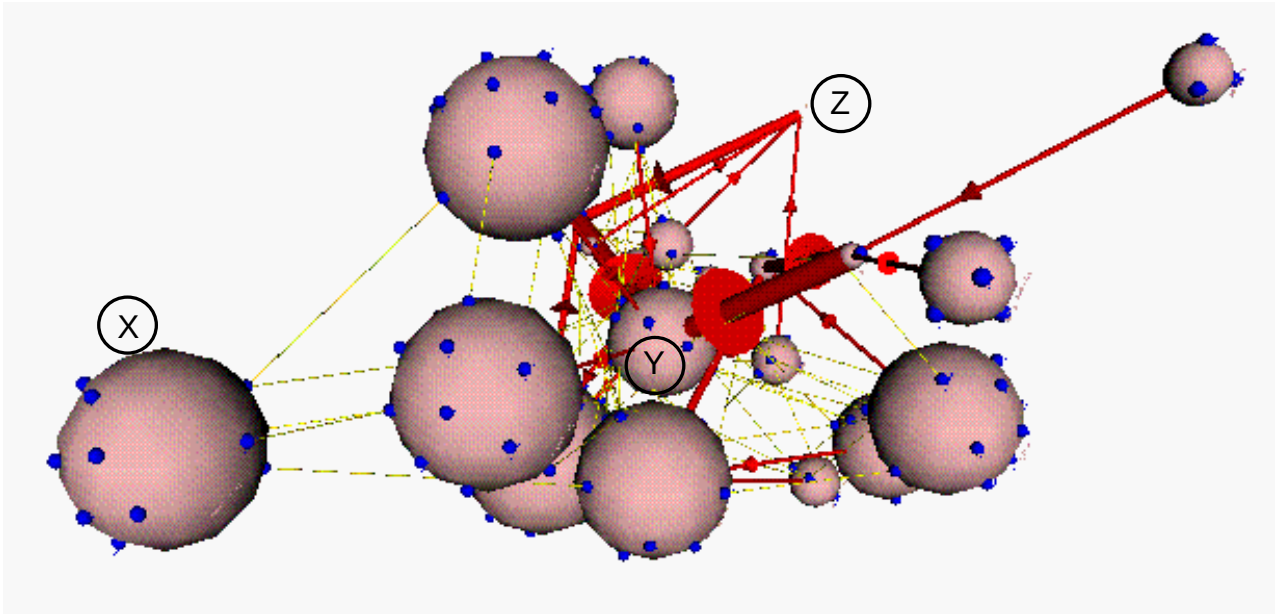

Figure 11: A class cluster for a Java package

selection region. Lines corresponding to three classes are highlighted, as are the points along each axis. Highlighted points on the scatterplot (B) and scatterplot matrix (C) views correspond to the same classes. This feature enables such exploratory tasks as investigation of classes which are outliers in one correlation to determine whether they are 'normal' or 'interesting' in other contexts. A typical investigation would involve brushing a set of points which are close to each other in one view (e.g. STMT, ν scatterplot) and observing whether they remain together in other views (as might be expected for a ν, defect density scatterplot) or diverge (as might be expected for views involving inheritance depth).

5 Class clusters

In this section, we discuss the design of a specific visualisation, class clusters, in terms of the design pipeline and the way the design was informed. Class clusters are discussed in greater detail elsewhere (Irwin & Churcher 2003, Churcher, Irwin & Cook 2004). A class cluster is shown in Figure 11.

The class labelled 'X' has no inheritance connection to the other classes. The class labelled 'Y' is the base class for the package shown and it appears at the center of the cluster. Class 'Z' is very small, indicating that it adds few methods or properties itself, but the thickness of the edge leading to its parent indicates that it nevertheless plays a part in the inheritance hierarchy since it has children of its own.

metrics selection A full range of metrics derived from the semantic model and associated parse trees was available. Metrics selected were ν and STMT (number of statements) for methods. For classes, WMC (number of public, non-constructor, locally defined methods) was computed and NLCS (the number of leaf classes supported) was derived from the direct inheritance data.

semantic relationships Encapsulation (*class-has-method*), invocation (*method-calls-method*) and inheritance (*class-extends-class*) were selected. Relationships involving all classes, but not interfaces, were included and only public, non-constructor, locally defined methods appear. Pre-layout filters extracted this data and transformed it into a format suitable for layout with ANGLE (Churcher & Creek 2001).

metaphor selection The primary objective was to indicate the 'neighbourhood' of a class within a package or system: physical proximity should reflect the strength of relationships between classes.

A secondary objective was to highlight the inheritance structure in order to separate structural from invocation system aspects.

design The metaphor was realised as a graph whose nodes correspond to classes and methods whose edges correspond to the semantic relationships identified earlier. Classes are represented by spherical nodes whose radii represent their WMC values; they are studded with smaller spheres representing their methods. Edges representing invocation connect these method nodes. An inhomogeneous force-directed layout algorithm was employed (Churcher et al. 2004) so that the inheritance edges would be the dominant contributors to the layout, while invocation edges play a secondary rôle. This causes the root of the inheritance tree to be at the centre of the cluster.

presentation The thicknesses of branches of 'real' trees are proportional to the mass of the other branches and leaves they support–and hence to their importance in the overall tree structure. Accordingly, post-layout mappings were used to map NLCS to inheritance edge thickness.

configuration XSLT transformations were used to generate VRML (Carey & Bell 1997) worlds for display in web browsers (see Figure 12).

An example of feedback in the design pipeline led to the revised visualisation shown in Figure 12. The revised design and presentation stages now involve using the geometry of nodes representing methods to reflect the values of method-level metrics ν and STMT. Instead of uniform spheres, the specific post-layout mappings now generate conical nodes for each method: $height \propto STMT$ and $base\_radius \propto \nu$. Outliers may now be readily identified.

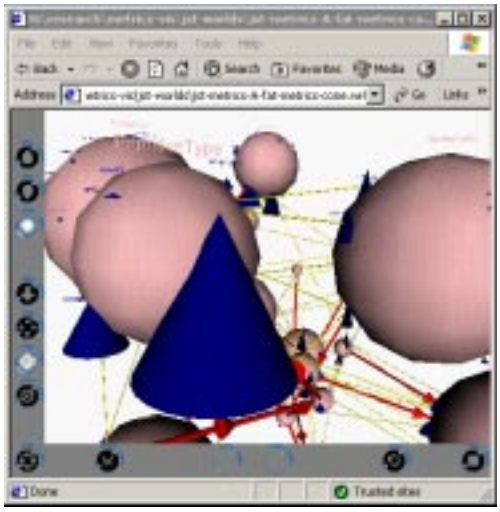

Figure 12: cluster with metrics

6 Conclusions & future work

In this paper, we have addressed the process of designing software visualisations and considered tools which can assist in this process. We have described the software visualisation design process in terms of a design pipeline. This is a conceptual pipeline parallel to the implementation pipeline responsible for actually generating the rendered visualisation from the initial data sets.

The stages of the design pipeline include:

- collating and evaluating available data, whether directly captured or indirectly computed.
- extracting the domain-specific semantic relationships present in the data sets.
- selecting a metaphor which both exploits the selected semantic relationships and addresses the primary goal of the visualisation.
- developing the design infrastructure of the visualisation including selection of factors such as layout algorithms, graph models and information visualisation techniques.
- refining the presentation details representing the mapping of domain properties to attributes of the rendered visualisation.

Tools such as SeeSoftLike and ggobi can inform this process in a variety of ways by assisting with the identification of relevant data and relationships.

The design pipeline helps ensure that a visualisation will be appropriate for its intended purpose, will be based on sound data and will convey effectively the information contained in the data. Without it, there is the risk that considerable effort will be spent on the development of a visualisation which lacks the sensitivity to distinguish values, fails to incorporate important relationships or conflicts with the prevailing domain models.

The outputs from the design pipeline primarily represent decisions, and artifacts derived from them, which influence the detail of the parallel implementation pipeline which actually delivers the rendered visualisation. They include such things as specific mappings from domain properties to attributes such as colour of geometry in the rendered visualisation.

Increased understanding of the design pipeline is valuable as we extend support for exploratory analysis through implementation pipeline flexibility.

This work is part of our ongoing software visualisation research. We are developing it further in a number of directions.

Firstly, we are assembling a library of resources and guidelines to assist visualisation designers by presenting suitable options and providing guidelines. Major components include:

Metaphor library to assist users to select an appropriate metaphor from a standard set and then tailor it to their particular applications. This would include a taxonomy based on perspectives such as those identified by Keown (Keown 2000).

Design library to assist with metaphor realisation. This would enable users to select from a range of information visualisation techniques, such as ways to represent hierarchical structures, suitable for representing their chosen metaphor. It would also include a range of ways to transform semantic relationships to geometric relationships

Filter library to support development of pre- and post-layout filters by providing templates suitable for extension.

In parallel, we are beginning to explore ways to provide software support for the design pipeline process. At the most trivial level this is little more than a checklist of principles and a list of resources. However, our longer-term goals include tools to allow the user to select increasingly specific metaphor, design and mapping aspects and to provide facilities to generate appropriate filter transformations and other elements to be used in the corresponding implementation pipeline. One challenge is how best to support user creativity—there can be a wide range of appropriate visualisations for a particular scenario. As our approach matures, we will be in a position to evaluate its effectiveness in realistic design situations where users other than ourselves are involved. Our approach will be to make use of heuristic evaluation techniques (Nielsen & Molich 1990, Nielsen 1992, Nielsen & Landauer 1993), in which small groups of evaluators seek violations of a given set of heuristics. Results suggest that these techniques can be very effective in detecting faults, thereby enabling them to be corrected earlier in the development cycle.

Among the aims of our current software visualisation research is the investigation of various heuristics for software engineering. Many of these, such as the law of Demeter (Lieberherr & Holland 1989), Riel's collection of heuristics for object oriented design or even design patterns (Gamma, Helm, Johnson & Vlissides 1995), are amenable to measurement and hence to visualisation.

Our approach, based on powerful parsing technology and driven by a sound semantic model, provides a sound basis for undertaking this programme.

References

Ball, T. & Eick, S. (1996), 'Software visualization in the large', *IEEE Computer* **29**(4), 33–43.

Carey, R. & Bell, G. (1997), *The Annotated VRML 2.0 Reference manual*, Addison-Wesley.

Chidamber, S. & Kemerer, C. (1994), 'A metrics suite for object oriented design', *IEEE Transactions on Software Engineering* **20**(6), 476–493.

Churcher, N. & Creek, A. (2001), Building virtual worlds with the big-bang model, *in* P. Eades & T. Pattison, eds, 'Information Visualisation 2001', Vol. 9 of *Conferences in Research and*

Practice in Information Technology, ACS, Sydney, Australia, pp. 87–94.

Churcher, N. & Shepperd, M. (1995), 'Towards a conceptual framework for OO software metrics', *ACM SIGSOFT Software Engineering Notes* **20**(2), 69–75.

Churcher, N., Irwin, W. & Cook, C. (2004), Inhomogeneous force-directed layout algorithms in the visualisation pipeline: From layouts to visualisations, *in* N. Churcher & C. Churcher, eds, 'InVis.au 2004: Proceedings of the Australasian Information Visualisation Symposium', Vol. 35 of *Conferences in Research and Practice in Information Technology*, ACS, Christchurch, New Zealand.

Churcher, N., Irwin, W. & Kriz, R. (2003), Visualising class cohesion with virtual worlds, *in* T. Pattison & B. Thomas, eds, 'Australasian Symposium on Information Visualisation, (invis.au'03)', Vol. 24 of *Conferences in Research and Practice in Information Technology*, ACS, Adelaide, Australia, pp. 89–97.

Churcher, N., Keown, L. & Irwin, W. (1999), Virtual worlds for software visualisation, *in* A. Quigley, ed., 'SoftVis99 Software Visualisation Workshop', University of Technology, Sydney, Australia, pp. 9–16.

Eick, S., Graves, T., Karr, A., Marron, J. & Mockus, A. (2001), 'Does code decay? assessing the evidence from change management data', *IEEE Transactions on Software Engineering* **27**(1), 1–12.

Eick, S., Steffen, J. & Jr., E. S. (1992), 'Seesoft—a tool for visualizing line oriented software statistics', *IEEE Transactions on Software Engineering* **18**(11), 957–968.

Everitt, B., Landau, S. & Leese, M. (2001), *Cluster Analysis*, 4th edn, Arnold.

Fagan, M. (1976), 'Design and code inspections to reduce errors in program development', *IBM Systems Journal* **15**(3), 182–211.

Fagan, M. (1986), 'Advances in software inspections', *IEEE Transactions on Software Engineering* **SE-12**(7), 744–51.

Fenton, N. & Pfleeger, S. L. (1997), *Software Metrics: A Rigorous & Practical Approach*, 2nd edn, International Thompson Computer Press.

Freeman, E. (1999), 'Building gargantuan software', *Scientific American Presents* **10**(4), 28–31.

Furnas, G. (1986), Generalised fisheye views, *in* 'Proc ACM SIGCHI '86 Conference on Human Factors in Computing Systems', pp. 16–23.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

ggobi (2004), 'GGobi Data Visualisation System Home Page', `http://www.ggobi.org`.

Henderson-Sellers, B. (1996), *Object-Oriented Metrics: Measures of Complexity*, Prentice Hall.

Ince, D. & Shepperd, M. (1992), *The Derivation and Validation of Software Metrics*, Oxford University Press.

Irwin, W. & Churcher, N. (2002), XML in the visualisation pipeline, *in* D. D. Feng, J. Jin, P. Eades & H. Yan, eds, 'Visualisation 2001', Vol. 11 of *Conferences in Research and Practice in Information Technology*, ACS, Sydney, Australia, pp. 59–68. Selected papers from 2001 Pan-Sydney Workshop on Visual Information Processing.

Irwin, W. & Churcher, N. (2003), Object oriented metrics: Precision tools and configurable visualisations, *in* 'METRICS2003: 9th IEEE Symposium on Software Metrics', IEEE Press, Sydney, Australia, pp. 112–123.

Johnson, B. & Shneiderman, B. (1991), Tree-maps: A space-filling approach to the visualization of hierarchical information structures, *in* G. Nielson & L. Rosenblum, eds, 'proc. Visialization '91', IEEE Computer Society Press, Los Alamitos, CA, pp. 284–291.

Keown, L. (2000), Virtual 3d worlds for enhanced software visualisation, Master's thesis, University of Canterbury, Department of Computer Science.

Lieberherr, K. & Holland, I. (1989), 'Assuring good style for object-oriented programs', *IEEE Software* pp. 38–48.

McCabe, T. J. (1976), 'A complexity measure', *IEEE Transactions on Software Engineering* **SE-2**, 308–319.

McCall, J., Richards, P. & Walters, G. (1977), Factors in software quality, Technical Report (RADC)-TR-77-369, Vols. 1–3, Rome Air Development Center, United States Air Force, Hanscom AFB, MA. Available as AD-A049-014, AD-A049-015 and AD-A049-055 from: NTIS, Springfield, VA.

Nielsen, J. (1992), Finding usability problems through heuristic evaluation, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 373–380.

Nielsen, J. & Landauer, T. K. (1993), A mathematical model of the finding of usability problems, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 206–213.

Nielsen, J. & Molich, R. (1990), Heuristic evaluation of user interfaces, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM Press, pp. 249–256.

O'Regan, G. (2002), *A Practical Approach to Software Quality*, Springer-Verlag.

Price, B., Baecker, R. & Small, I. (1993), 'A principled taxonomy of sofware visualization', *Journal of Visual Languages and Computing* **4**(3), 211–266.

Robertson, G., Mackinley, J. & Card, S. (1991), Cone trees: Animated 3d visualizations of hierarchical information, *in* 'Proc. ACMSIGCHI '91 Conf. on Human Factors in Computing Systems', New Orleans, Louisiana, pp. 189–194.

Sarkar, M. & Brown, M. (1994), 'Graphical fisheye views', *Communications of the ACM* **37**(12), 73–84.

Schroeder, W., Martin, K. & Lorensen, B. (1998), *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, 2nd edn, Prentice Hall.

together (2004), 'Borland Together IDE Home Page', `http://www.borland.com/together`.